1983

Hardware and software implementation of an interface between the unibus and general purpose interface bus

Blocher, Ayers Haden

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Hardware and Software Implementation
of an Interface Between
the Unibus and the General Purpose
Interface Bus

by

Ayers Haden Blocher III

March 1983

Thesis Advisor: Kenneth Gray

Approved for public release, distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Hardware and Software Implementation of an Interface Between the Unibus and the General Purpose Interface Bus		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis March 1983 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ayers Haden Blocher III		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE March 1983 13. NUMBER OF PAGES 68
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

GPIB; HPIB; Interface; Unibus; Computer; PDP-11; DR11-C; Satellite

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Satellite Communications Laboratory at the Naval Postgraduate School uses a PDP-11/34A minicomputer to develop software in support of a satellite signal monitoring system. The General Purpose Interface Bus (GPIB) interconnects several general measurement devices used in support of the laboratory. The laboratory uses these measurement devices for diagnostic and simulation tests related to research in the satellite signal monitoring field. This thesis discusses the development of the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

hardware and software interface between the PDP-11/34A Unibus and the GPIB. The interface permits high level language programs under the control of the Unix operating system (version 6) on the PDP-11/34A to access any device on the GPIB.

Approved for public release, distribution unlimited.

Hardware and Software Implementation of an Interface Between the Unibus and the General Purpose Interface Bus

by

Ayers Haden Blocher III
Lieutenant Commander, United States Navy
B.S., University of Missouri at Rolla, 1972

Submitted in partial fullfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March, 1983

ABSTRACT

The Satellite Communications Laboratory at the Naval Postgraduate School uses a PDP-11/34A minicomputer to develop software in support of a satellite signal monitoring system. The General Purpose Interface Bus (GPIB) interconnects several general measurement devices used in support of the laboratory. The laboratory uses these measurement devices for diagnostic and simulation tests related to research in the satellite signal monitoring field. This thesis discusses the development of the hardware and software interface between the PDP-11/34A Unibus and the GPIB. The interface permits high level language programs under the control of the Unix operating system (version 6) on the PDP-11/34A to access any device on the GPIB.

TABLE OF CONTENTS

I. INTRODUCTION---------- 9

A. RATIONALE FOR INTERFACE DEVELOPMENT---------- 10

B. INTERFACE DESIGN DEVELOPMENT---------- 12

II. UNIBUS AND GPIB CHARACTERISTICS---------- 13

A. UNIBUS---------- 13

B. GPIB---------- 13

1. Data Transfer---------- 16

2. Handshaking---------- 16

C. SUMMARY---------- 19

III. HARDWARE IMPLEMENTATION---------- 20

A. DR11-C INTERFACE---------- 22

1. Registers---------- 22

B. CCI BUFFER/DRIVER BOARD---------- 23

1. Connectors---------- 23

2. Signal Inversion and Feedback---------- 27

3. Wiring System---------- 29

C. SUMMARY OF HARDWARE IMPLEMENTATION---------- 32

IV. UNIX OPERATING SYSTEM---------- 33

A. GENERAL---------- 33

B. UNIX I/O---------- 34

1. Device Drivers---------- 34

2. I/O Support Software---------- 36

C. SYSTEM RECONFIGURATION---------- 41

1. Driver Installation---------- 42

2. Driver Modification---------- 45

V. SOFTWARE IMPLEMEMTATION---------- 47

A. GPIB INTERFACE DRIVER---------- 48

1. Dropen()---------- 48

2. Drclose()---------- 49

3. Drwrite()---------- 49

4. Drread()---------- 51

5. Drint()---------- 53

B. GPIB DRIVER SUPPORT SOFTWARE---------- 54

1. Ieout.h---------- 56

2. Iein.h---------- 57

C. SUMMARY---------- 58

VI. CONCLUSIONS---------- 61

A. ADVANTAGES---------- 61

B. DISADVANTAGES---------- 62

C. COMPARISON WITH A COMMERCIAL PRODUCT---------- 63

D. RECOMMENDATIONS---------- 65

LIST OF REFERENCES---------- 67

INITIAL DISTRIBUTION LIST---------- 68

LIST OF TABLES

II.1 GPIB Signal Descriptions---------- 15

III.1 Interface Connector Pin Assignments---------- 26

III.2 CCI Component Pin Assignments---------- 31

LIST OF FIGURES

1.1 PDP-11 Instrumentation System---------- 11

2.1 GPIB Handshake Sequence---------- 18

3.1 GPIB interface Component Block Diagram---------- 21

3.2 DR11-C Register Addresses---------- 24

3.3 Interface System Data Flow---------- 25

3.4 DR11-C Register Bit Assignments---------- 28

3.5 Condensed CCI Wiring Schematic---------- 30

4.1 Unix I/O File Tree---------- 38

5.1 Sample Program To Write To a GPIB Device---------- 55

5.2 Sample Program To Read From a GPIB Device---------- 59

I. INTRODUCTION

The Satellite Communications (SATCOM) laboratory at the Naval Postgraduate School utilizes a PDP-11/34A minicomputer as a research tool for satellite signal analysis. The General Purpose Interface Bus (GPIB) interconnects instrumentation used as simulation and diagnostic tools for the signal analyzer's computer software. Under usual circumstances a small desk top computer such as the HP-9825 and the attached computer interface card (HP-98034A) control the flow of signals along the GPIB. In this configuration the GPIB devices make up a separate system from the PDP-11/34A controlled system. Bringing the GPIB devices under control of the PDP-11/34A requires a hardware and software interface between the minicomputer Unibus and the GPIB. The interface eliminates the need for the HP-9825 and HP-98034A. The following sections discuss the characteristics of the hardware and software interface between the two bus systems.

There are six main sections in this thesis. The remainder of the introduction explains the rationale for developing the interface and discusses development design constraints. The next section discusses the characteristics of the Unibus and the GPIB. The following section treats the hardware implementation and physical structure of the interface in concert with the Unibus and the GPIB. The Unix operating system is then discussed, especially in the areas of I/O and system reconfiguration. The following section deals with with the software required to drive the hardware under the constraints and requirements of

the Unix operating system. The conclusion summarizes the hardware and software characteristics of the interface and suggests possible alternatives and amendments to the existing system. The conclusion also provides a comparison of the developed interface with a commercially available product.

A. RATIONALE FOR INTERFACE DEVELOPMENT

With no interface in place the PDP-11/34A system and the simulation/diagnostic device system are independent. The PDP-11/34A has no way to communicate with any of the GPIB equipment, and an independent computer (e.g., the HP-9825) controls the the GPIB devices. Additionally, the simulation/diagnostic system is constrained to operate under the limited flexibility of the desk top computer software. With the interface in place an operator may control all the devices on the GPIB from a console on the PDP-11/34A system. The operator may write programs under the UNIX operating system using the higher level C language and, treating the GPIB as just another peripheral device, send or receive data over the GPIB.

In summary there are three advantages to interfacing the two systems:

1. The need for the desk top computer is deleted.
2. One Unibus console controls both systems.
3. Unix operating system software drives the GPIB.

Figure 1.1 is a block diagram of an interfaced system to illustrate the effect of the GPIB interface.

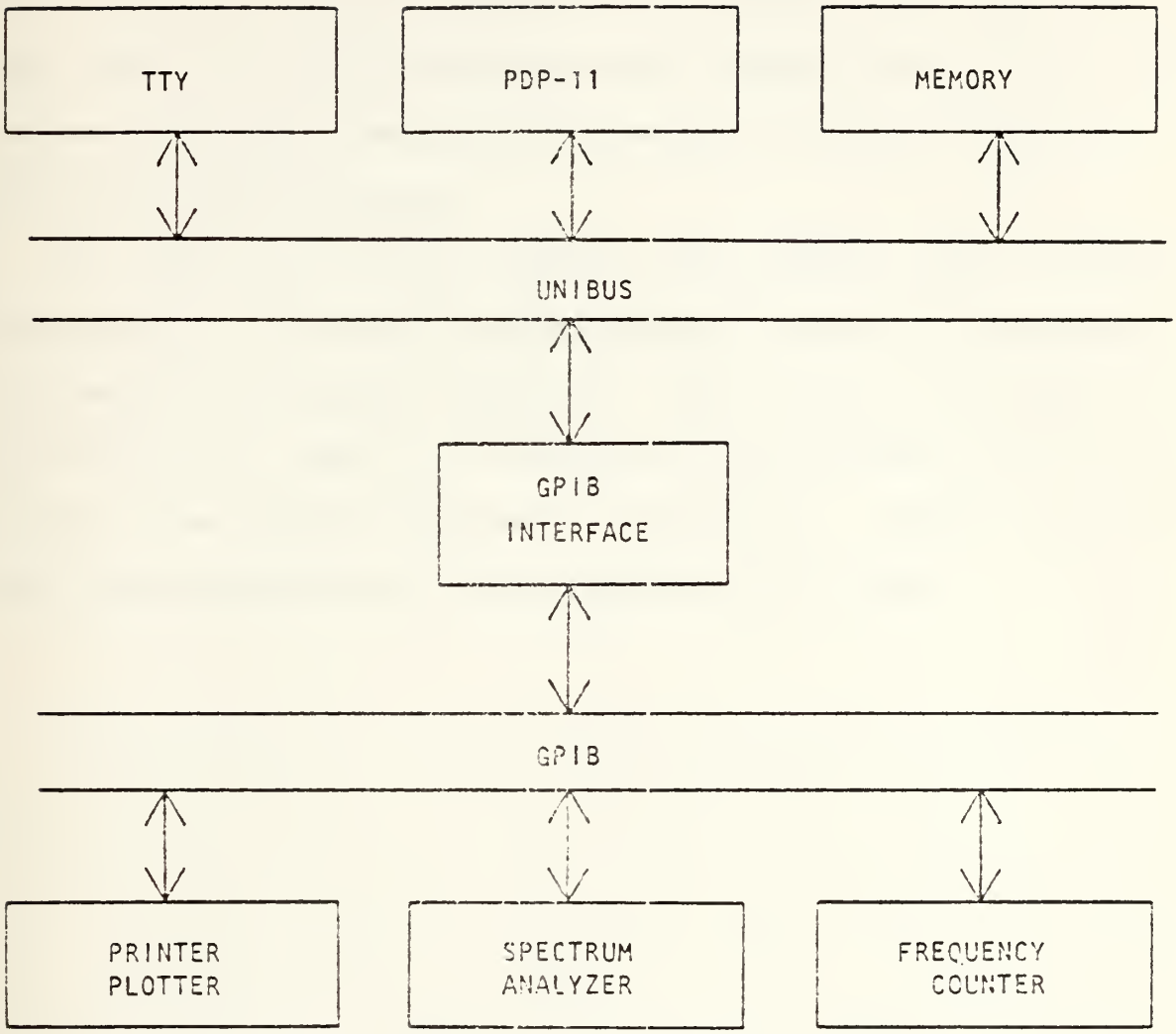


Fig 1.1 PDP-11 Instrumentation System

B. INTERFACE DESIGN DEVELOPMENT

The design of the GPIB interface is subject to two major constraints. First, the hardware for the interface must not take up more than one slot within the PDP-11/34A mainframe. In other words, total board size must not exceed a single hex-height card. Second, the software to interface the systems must conform to the I/O requirements of the Unix operating system.

The following sections discuss the hardware and software implementation of the GPIB interface based on the needs and constraints discussed in the preceding paragraphs. The hardware development is discussed first since it is independent of the operating system of the computer. The software development is preceded by a treatment of the Unix operating system and in particular the Unix I/O area.

II. UNIBUS AND GPIB CHARACTERISTICS

The hardware and software interface implementation connects two independent bus systems, Unibus and GPIB. Discussion of the Unibus is limited to a brief description since no hardware or software changes are required on the DR11-C General Purpose Interface module, which initially interfaces the Unibus, beyond address wire jumper connections. Digital Equipment Corporation supplies information on the detailed operation and characteristics of the Unibus [Ref. 1: pp. 2-20]. The discussion of the GPIB is more descriptive since the data transfer and handshaking requirements of that system directly govern the design of the hardware and software needed to connect the DR11-C and the GPIB.

A. UNIBUS

The PDP-11 Unibus is composed of 56 bidirectional lines which carry signals among devices on the bus (CPU, memory, I/O). Of these 56 lines, the DR11-C uses 45 [Ref. 2: pp. 8-9]. These signals and their respective functions are in table 4 of the DR11-C Instruction Manual [Ibid.]. A thorough analysis of the Unibus operation and signal characteristics is available in the support literature.

B. GPIB

The General Purpose Interface Bus (GPIB) is a 16 line bidirectional bus which meets the requirements of the IEEE-488-1975 standard. It is designed to facilitate the exchange of data among devices connected to

the bus. The Hewlett-Packard implementation of the GPIB is called the HP-IB, and thus the terms GPIB and HP-IB have interchangeable meanings.

This section contains only a limited discussion of GPIB characteristics. Reference material [Ref. 3 and Ref. 4] concerned with the analysis of the bus is available which provides a thorough treatment of the GPIB signals and capabilities. Table II.1 describes the signal lines on the GPIB which control activity on the bus. In addition to those listed, there are eight lines designated to carry data among devices on the bus. As seen in Table II.1, all signals on the GPIB are negative-true polarity. That is, a high signal means false, and a low signal means true. In order to ensure clarity, subsequent discussion of the term "DATA" (all capitals) refers to the eight lines which carry information between devices, while "data" refers to the 16 bit contents of the bus or a register in the system.

One device on the bus is designated as the active controller. The active controller issues instructions to other devices and controls all the traffic on the bus. Any device may be designated as an active controller, but this job is usually assigned to a computer or calculator.

One device on the bus is designated as the system controller. The system controller is established through hardware connections and cannot be passed to another device on the bus. The system controller automatically becomes the active controller when power is turned on or the bus is reset.

ATN: Attention is driven by the active controller and indicates whether address commands (ATN low) or data (ATN high) are transmitted.

IFC: Interface Clear is used only by the system controller to initialize the bus via the abort message. When IFC is low for at least 100 microseconds, all talkers and listeners are stopped, the serial poll mode is disabled, and control is returned to the system controller. When IFC is high, it has no effect on the bus operation.

SRQ: Service Request is driven low by a device to indicate that it wants the attention of the controller. SRQ may be set low at any time except when IFC is low.

EOI: End or Identify may be used to indicate the end of an instruments character string. When ATN is high, the addressed talker may indicate the end of its data by setting EOI low at the same time that it places the last byte on the data lines.

REN: Remote Enable is driven by the system controller and is one of the conditions for operating instruments under remote control. Only instruments capable of remote operation use REN and they monitor it at all times. Instruments that do not use REN terminate the line in a resistor load. The system controller may change the state of REN at any time.

NRFD: Not Ready For Data indicates that all listeners are ready to accept information on the data lines. When NRFD is low, one or more listeners are not ready for data.

NDAC: Not Data Accepted is high to indicate the acceptance of information on the data lines by all listeners. When NDAC is low, all listeners have not accepted the information.

DAV: Data Valid indicates the validity of information on the data lines. When DAV is low, the information on the data lines is valid for the listener(s). When DAV is high, the information on the data lines is not valid.

Table II.1 GPIB Signal Descriptions

1. Data Transfer

Though there are eight lines available for DATA (D1 through D8) most of the GPIB instruments base their DATA on the seven bet ascii code. DATA is transferred ascii character at a time, byte serial and bit parallel. For the GPIB interface all DATA is presumed to be ascii coded. Therefore, only seven DATA bits are implemented on the GPIB interface.

Transfer of data, whether in a read or write context, is initiated by the active controller which establishes the talker/listener relationship among the devices on the bus. The active controller sends the talker device number and the listener device number over the DATA lines with the ATN signal true. While ATN is true, the devices on the bus interpret DATA as bus control messages rather than actual data messages. Once the talker and listener are designated the active controller sets ATN false which signals the talker to begin sending its data message over the DATA lines. Designation of the talker and listener is usually preceded by an UNLISTEN bus command from the computer which effectively resets the bus for DATA flow.

2. Handshaking

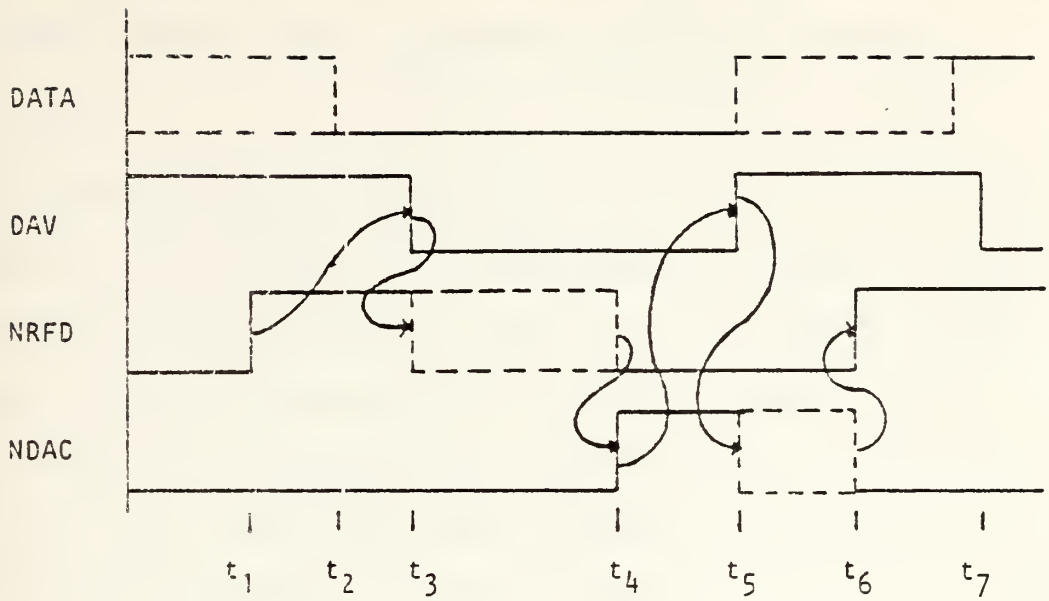
Handshaking is the term used to describe the process whereby a controlling device and a peripheral device talk to one another during an exhange of data. In the GPIB handshaking process the talker notifies a listener of available DATA. The listener signals both readiness for DATA and completion of DATA processing. The process on the GPIB is asynchronous, and no restrictions are placed on the data rates of any instrumentation on the bus. Because of the asynchronous nature of the

DATA exchange and handshaking process, the slowest device on the the GPIB (e.g., a printer/plotter) controls the time required to complete the entire procedure.

Handshaking is under the control of three signals on the bus: DAV, NRFD, and NDAC. The talker controls the DATA lines and DAV. The listeners control NRFD and NDAC. The following description of the handshaking process is illustrated in Figure 2.1. The process is the same for bus commands as well as for data message transfer.

DATA transfer is initiated by all the listeners on the bus by setting NRFD high. This signifies that they are ready for DATA (Not Ready For Data is false). When the talker senses that the NRFD line is high it places DATA on the DATA lines and sets the DAV line low (Data Available is true). When the addressed listener senses that DAV is low it takes in the data, processes it and signifies when processing is over by setting NDAC high (Not Data Accepted is false) which tells the talker that the DATA has been accepted and it no longer need be held on the DATA lines. When the talker senses NDAC is high it sets DAV high (false) while it places the next DATA byte on the DATA lines. When DAV is sensed as high by the listener, it sets the NDAC line back low then NRFD high to start the cycle over. Note that both NRFD and NDAC cannot be high at the same time. Such a state is illegal.

The assertive state of NDAC and NRFD is high. Since all the listeners have their repective bus lines tied together, all listeners must set their corresponding signals high before that line on the GPIB goes high. This is the wired AND situation which allows the talker to recognize when the slowest device has taken the DATA and is ready for



t_1: Listener ready for data

t_2: Data placed on data lines by talker

t_3: Data on data lines valid

t_4: Data accepted by listener

t_5: Data no longer valid and may be changed by talker

t_6: Listener ready for new data

t_7: Cycle repeats

Note: Curved arrows indicate interlocked signal sequence.

Fig 2.1 GPIB Handshake Sequence

more. All listeners on the bus respond to the talker. Only the addressed listener, however, processes that DATA as a message.

C. SUMMARY

The General Purpose Interface Bus (GPIB) is a 16 line bus system intended for use primarily with instrumentation utilizing the seven bit ascii code for data transfer in a serial byte and parallel bit mode. The Unibus is a 56 line system which is driven by the PDP-11 CPU. In order for the PDP-11 to communicate with a device on the GPIB the data must pass over the Unibus data lines, through the GPIB interface, onto the GPIB, and into the device. As will be seen in section III, the DR11-C handles the exchange of data and handshaking signals from the Unibus without modification of the DR11-C module or software assistance. However, exhange of data between the GPIB and the DR11-C and then onto the desired device requires additional hardware and software support.

III. HARDWARE IMPLEMENTATION

The GPIB interface hardware relationships between the Unibus and the GPIB are illustrated in the block diagram in Figure 3.1. The hardware for the interface consists of one DR11-C general purpose interface module (hereafter referred to as the DR11-C), and a locally designed and constructed buffer module, called the CCI. The DR11-C was chosen to intially interface the Unibus since MDB Incorportated, manufacturer of the DR11-C, specifically designed it to act "as an interface to transfer data between a Digital Equipment Corporation PDP-11 Unibus and the user's peripheral device." [Ref 2: p.1] The DR11-C is therefore specifically designed to transfer 16 data bits. In the following discussion the terms "data" and "data bits" refer to the 16 elements of a word in the PDP system. When referring to the specific elements of the GPIB which carry the ascii character (the seven least significant bits) the term "DATA" is used. Since the GPIB is a 16 line bidirectional bus, the DR11-C is well suited to act as the initial transfer medium for the interface if the handshake/bus command lines of the GPIB are treated in software as data bits. The CCI board is required to act as a buffer between the two distinct (input and output) registers on the DR11-C and the single bidirectional GPIB connector.

This section describes the GPIB interface hardware: DR11-C and CCI. The discussion of the DR11-C is limited to a functional description since the DR11-C instruction manual describes the interface board in detail, providing figures and schematics to assist in future DR11-C

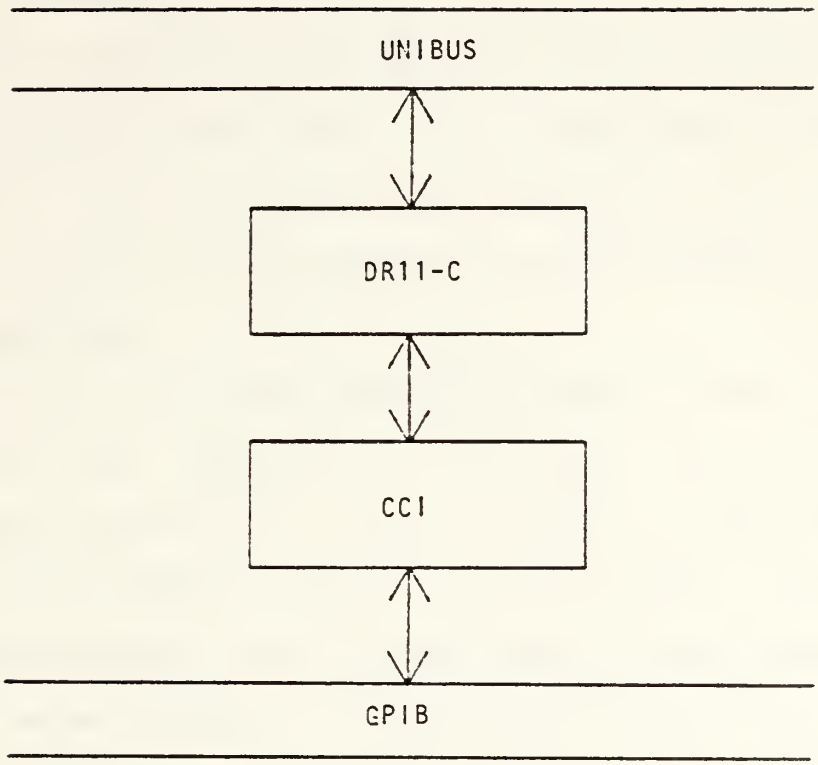


Fig 3.1 GPIB Interface Component Block Diagram

implementation. The CCI board is discussed in greater detail since the available documentation on it is limited.

A. DR11-C INTERFACE

The DR11-C board functionally accepts 16 data bits from the Unibus, sends 16 data bits to the Unibus, and performs necessary handshaking in the process. A full description of Unibus/DR11-C data transfer is available in pages 10-12 of the DR11-C Instruction Manual.

1. Registers

The DR11-C has three registers: control/status, output, and input. All three registers are 16 bits wide, but the control/status register uses only six bits of the 16. The board is wired such that the address of the output register is two locations greater than the control/status register, and the input register is two locations greater than the output register. In other words, the three registers occupy consecutive 16 bit memory locations in the PDP-11 memory space.

The DR11-C is manufactured with the control/status register wired to address 767770 (all addresses are written in octal). This address may be changed by wired jumper as discussed on page 7 of the DR11-C Instruction Manual. The address of the control/status register is set to address 767730 for use in the SATCOM laboratory. This was done in accordance with the instructions on page 15 of the DR11-C Manual which requires that the address of the control/status register on the DR11-C be in concert with the associated vector address [Ref. 2: p. 15]. The vector address must be chosen such that it is higher than the vector address of any KL-11 in the system. Unix makes use of the KL-11 drivers for the teletype consoles. To assign the vector address for the DR11-C

one must check the Unix software file 'l.s' which contains the vector address assignments for the system (Unix filenames appear in single quotes). From this file, the KL-11's occupy vector addresses 300 through 330. Therefore, we assign the DR11-C the vector pair 340,344. The corresponding control/status register address is 747730 [Ibid.].

Figure 3.2 illustrates the relationship among the three DR11-C registers and their memory locations. In the PDP-11 memory structure each word is made up of a high and low byte of eight bits. The register location is identified by the low (even numbered) byte address. For example, the control/status register of the DR11-C is identified with address 767730 but includes 767731.

B. CCI BUFFER/DRIVER BOARD

The CCI board is a locally constructed interface designed to resolve the connector mismatch between the DR11-C and the GPIB, provide logic reversal for the negative-true GPIB from the positive-true DR11-C, plus provide a means for the output register of the DR11-C to feed to its input register. These features are discussed in the following paragraphs with a description of the CCI wiring system.

1. Connectors

The CCI board has three connectors: Two of these connect to the input and output register connectors of the DR11-C board, and one connects to the bidirectional GPIB. Figure 3.3 depicts the physical makeup of the connectors and their designation together with the data flow through the interface.

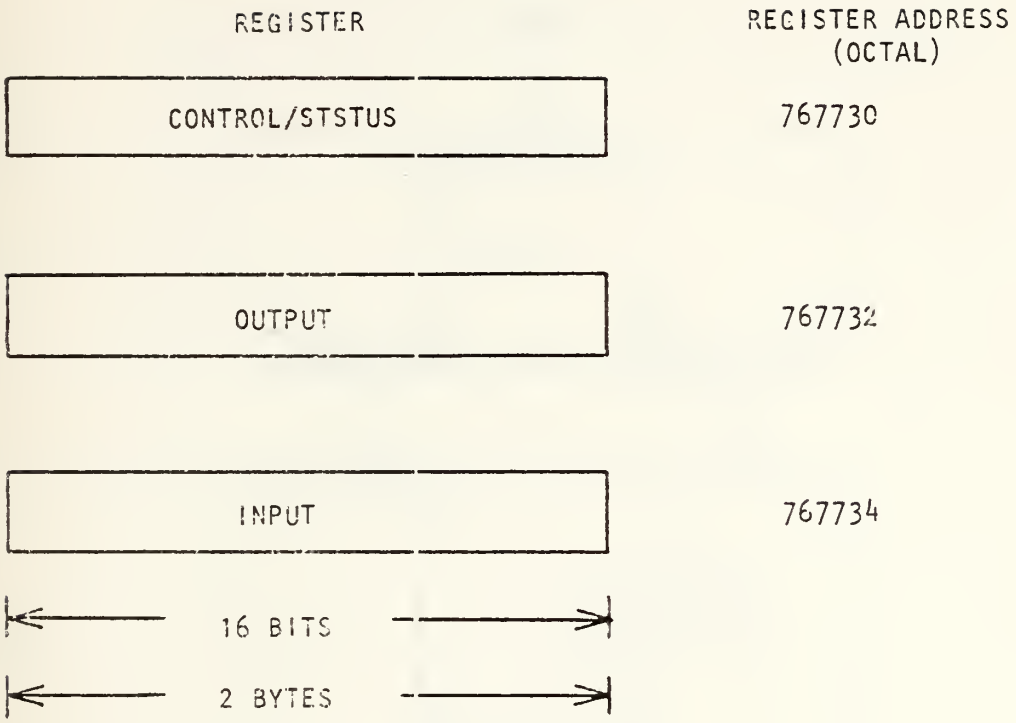
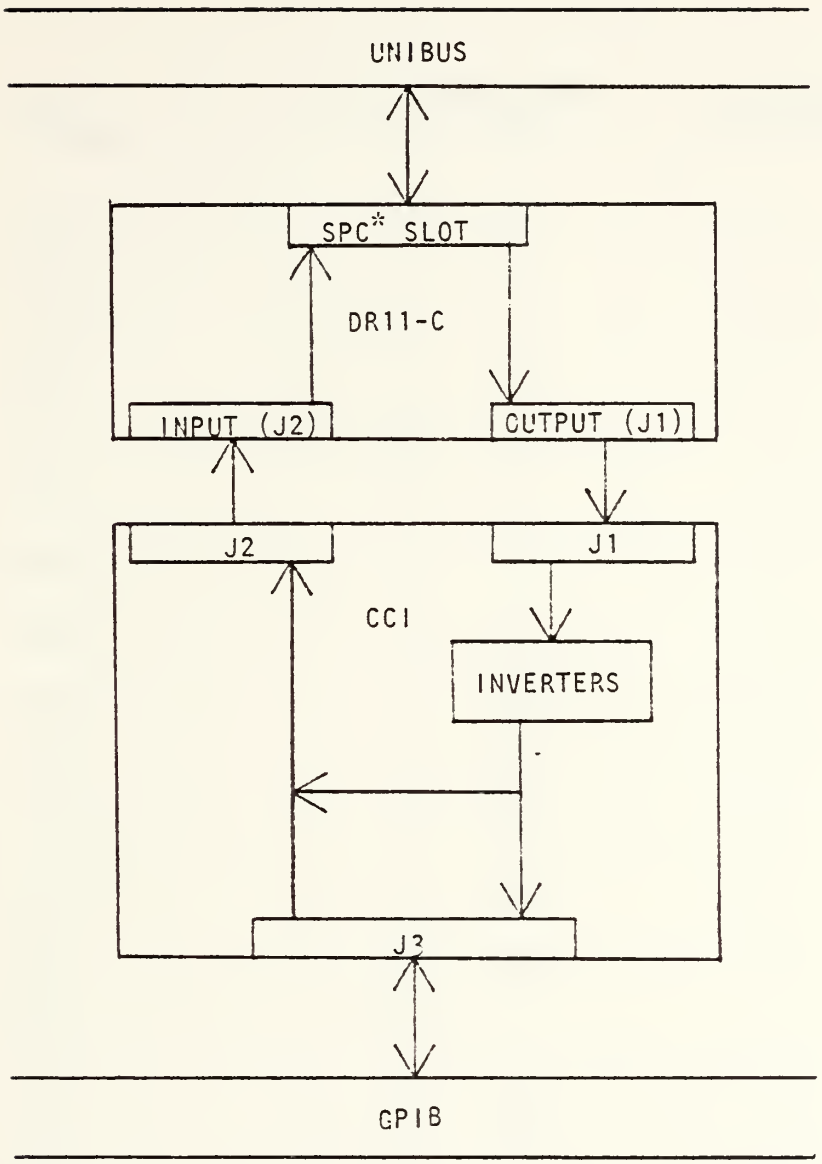


Fig 3.2 DR11-C Register Addresses



*SPC: Small Peripheral Slot on PDP-11 mainframe

Fig 3.3 Interface System Data Flow

GPIB SIGNAL	DR11-C CONNECTORS J1	J2	CCI CONNECTORS J1 and J2	J3
NRFD	JJ	M	11	13
NDAC	HH	N	12	15
EOI	FF	P	13	9
IFC	BB	V	17	17
SRQ	AA	W	18	19
REN	Z	Y	19	10
DAV	W	Z	22	11
ATN	T	CC	25	21
D7	R	EE	27	6
D6	N	HH	29	4
D5	L	KK	31	2
D4	U	BB	24	7
D3	NN	H	7	5
D2	K	LL	32	3
D1	C	T	38	1

Table III.1 Interface Connector Pin Assignments

Since the DR11-C I/O registers are 16 bits wide and the GPIB is a 16 line bus, a one-to-one correspondence between the registers and the GPIB is used. That is, each line of the GPIB feeds into one bit position of the input and output registers of the DR11-C. The bit assignments of the DR11-C registers are shown in Figure 3.4. The ground line from the GPIB does not have a partner connection on the DR11-C registers, since the CCI module provides grounding to the computer mainframe. Therefore, one bit position on those registers is not used. The bit assignment for the input register is identical to that of the output. The assignment of bit positions is arranged so that the lower byte of each register contains the DATA (plus the ATN bit), and the higher byte contains the bus control/handshaking bits.

The GPIB is wired to both the input and output registers on the DR11-C so that it may receive DATA and bus commands from the output register and supply DATA and bus commands to the input register without the use of a switch boards. Table III.1 shows the pin designations for the connectors shown in Figure 3.3.

2. Signal Inversion And Feedback

The GPIB handshaking process requires that the handshake signals on the bus be monitored as they are toggled by the talker and listener. The Unibus will always be a either a talker or a listener when the interface is in use, and thus requires a register to monitor GPIB handshake activity. The register used for this purpose is the DR11-C input register. Therefore, both the GPIB and the DR11-C output register must have means to transfer data to the input register.

GPIB SIGNAL	NRFD	NDAC	EOI	IFC	SRQ	REN		DAV	ATN	D7	D6	D5	D4	D3	D2	D1
Bit #	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Note: Input and output register bit assignments are identical.

Fig 3.4 DR11-C Register Bit Assignments

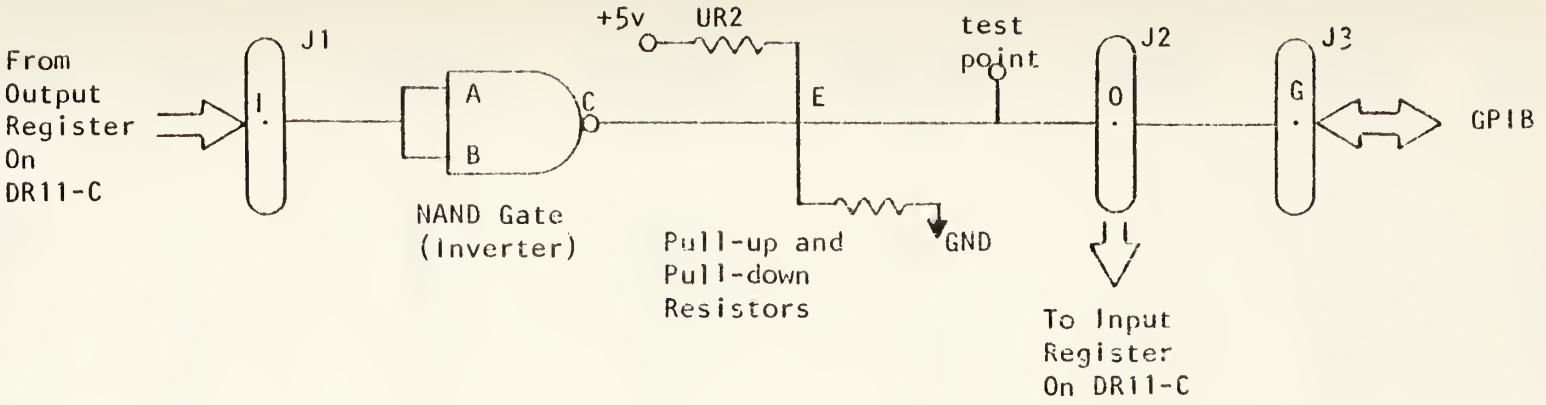
However, when the GPIB transfers data the DR11-C output register must be protected against having its contents altered. The inverters prevent the undesirable feedback from the GPIB to the output register while conveniently inverting the positive-true contents of the output register to the negative-true GPIB (see Figure 3.3).

There exists an inconvenience in the CCI board as it is currently implemented. The data returned to the input register is the inverse of that in the output register. Although inconvenient this is corrected in the software of the driver routine. In view of the size limitation of the CCI board, the software method of resolving the polarity difference is more practical than wiring on another set of inverters to the single height board.

3. Wiring System

Figure 3.5 is a condensed schmatic of the wiring system of the CCI board. A full schematic [Ref. 5] is available in the SATCOM lab. Data flow through all lines, except the REN line, are identical and are represented by one line labled with the indicated letters at pinout positions. Table III.2 lists the relationsnip between each line and its associated inverter chip, resistor pin, connector designations, and pin numbers.

The REN line has one additional inverter in its line to provide the user with a default position of true rather than false. Two inverters are provided so that the REN line may be made to default to false through bypassing one of the inverters with a hardwire change. This eliminates the need to change software in the driver routine or install a new inverter in the CCI.



Notes:

1. The CCI board has four 14-pin 7438 open collector NAND chips designated U1, U2, U3, and U4. Each chip contains four NAND gates. Pin 14 on each chip is connected in parallel to 5 volts and through a 0.01 microfarad capacitor to ground. Pin 7 is connected to ground.

2. Pull-up and Pull-down resistors are contained in 16 pin DIP resistor networks.
   a) Pull-up: 33 kilohms, 2% tolerance. Pin 16 connected to +5 volts.
   b) Pull-down: 62 kilohms, 2% tolerance. Pin 16 connected to ground.

Fig 3.5 Condensed CCI Wiring Schematic

GPIB SIGNAL	CCI J1 PIN (I)	NAND GATE DESIGNATOR	NAND GATE INPUT PINS (A,B)	NAND GATE OUTPUT PIN (C)	RESISTOR PIN (E)	CCI J2 PIN (O)	CCI J3 PIN (G)
NRFD	11	U2	9,10	8	2	11	13
NDAC	12	U2	1,2	3	3	12	15
EOI	13	U3	12,13	11	4	13	9
IFC	17	U2	12,13	11	5	17	17
SRQ	18	U3	9,10	8	6	18	19
REN	24	U3	4,5	6	7	24	7
		U4	9,10	8			
DAV	22	U3	1,2	3	8	22	11
ATN	25	U1	1,2	3	10	25	21
D7	27	U1	4,5	6	11	27	6
D6	29	U1	9,10	8	12	29	4
D5	31	U1	12,13	11	13	31	2
D4	24	U4	12,13	11	9	24	7
D3	7	U2	4,5	6	1	7	5
D2	32	U4	4,5	6	14	32	3
D1	38	U4	1,2	3	15	38	1

Table III.2 CCI Component Pin Assignments

C. SUMMARY OF HARDWARE IMPLEMENTATION

The DR11-C and the CCI buffer/driver boards act as a single interface card for transfer of data between the Unibus and GPIB. The design of the interface is provided for software which treats the handshaking/bus command lines in the same way as DATA. This design has no hardwire connections to the control/status register of the DR11-C. Therefore, transfer of data is independent of the interrupt system. All bit checking and DATA transfer is accomplished through manipulation of the DR11-C output and input registers. The input register serves dual duty as a register to read DATA from the GPIB and to provide a location to monitor the status of handshaking/bus command lines on the GPIB. Prior to the discussion of the software to support the above hardware implementation, the Unix operating system is thoroughly discussed to provide the system requirements of I/O driver software.

IV. UNIX OPERATING SYSTEM

This section contains a general discussion of the Unix operating system as installed in the SATCOM laboratory. Attention is focused on the I/O area and the procedure for reconfiguration of the operating system when a change is made to the I/O drivers and support software.

A. GENERAL

The Unix operating system is a multi-user, interactive operating system. A full assortment of documentation and manuals are available which fully describe the system. These are primarily the work of Dennis M. Ritchie, Ken Thompson, and Brian W. Kernighan of Bell Laboratories who developed the system through a series of versions beginning in 1969. For the purposes of this report the user is presumed to have a working knowledge of Unix or has access to the support manuals.

The most significant feature of Unix which impacts on the interface between the Unibus and GPIB is that Unix treats virtually everything as a file. Text, programs, functions, and even peripheral devices are all associated with a file. The reader is referred to the in depth treatment of the Unix file system by Ritchie and Thompson [Ref. 6] if any restructuring of the operating system is planned.

Also significant is the fact that Unix is dominated by use of the C programming language. Some files are written in the Unix assembly language (see Ref. 7 for details of the Unix assembler) but the vast majority are written in C. There is no attempt here to fully discuss the C programming language. Ritchie [Ref. 8 and Ref. 9] provides a very

readable treatment of C in a tutorial format. Again, the reader is presumed to have a working knowledge of the C language or access to the referenced manuals.

B. UNIX I/O

In order to implement a hardware and software interface between the Unibus and GPIB, a thorough understanding of the Unix I/O system is required. Though most of the following information is located in one or more of the manuals listed in the bibliography or list of References, there is no one source to which a user may turn without sifting through a plethora of unnecessay text. More importantly there are subtle differences in the system installed in the SATCOM laboratory which are not reflected in the support manuals. These differences will be noted as they occur in this discussion. Additionally, references to Unix I/O devote much effort to distinguish between transfer of a block of information and the transfer of a single character. This work will only deal with character transfer since the GPIB interface is equipped only to handle one character at a time.

1. Device Drivers

Every I/O device on the Unibus is associated with a device driver (sometimes called a handler) which interfaces the Unibus with the device. The driver contains a minimum of four routines if it is capable of both reading and writing. These routines are open, close, read, and write. The contents of each routine varies depending on the device, but their purpose is common to all devices. That is, they supply the Unix calls of the same name with the instructions necessary to cause a satisfactory interfacing between the device and the operating system.

Because of their importance to the structure of the device driver, each of the open, close, read, and write calls are briefly discussed in the following paragraphs.

The open call has the form: open(arg1,arg2). Open is called each time a file is to be read from or written to. Arg1 is the full device name and arg2 is 0,1, or 2 depending if the file is to read from (0), written to (1), or both (2). The file is the driver containing the read and write routines. The open routine acts as a sentry to allow only one user to access the drivers read and/or write routine at any one time. For instance, if user A desires to write on a paper punch the open routine checks to ensure no one else is currently using that paper punch. If user B has already accessed the punch and is occupying the machine at the time of A's request, the open routine returns an I/O error to user A. If user B is not using the punch the open routine returns a single digit integer called a file descriptor which is used in subsequent calls.

The close call has the form: close(fd). The close routine is the partner of the open routine. When the user is finished reading from or writing to the file, the close routine takes the file descriptor returned by open and resets the driver for access by another user. The close routine is called once for every call on open.

The read call has the form: read(fd,buf,nbytes). The read call uses the file descriptor returned by a successful open call to identify which read routine will be used to process the read request. The 'fd' notation is commonly used throughout the Unix reference manuals to denote the integer value for the file descriptor. 'Buf' is the name of

a buffer into which read data is placed. 'Nbytes' is the number of characters to be placed in the buffer.

The write call has the form: write(fd,buf,nbytes). The write call performs in the same manner as the read. The only difference is that the read call may not actually read as many as 'nbytes' characters. If the device supplies only 10 characters, 10 will be read, even if 100 are requested. The write call considers the number of characters to be transferred an order, not a request.

Each device capable of being read from and being written to must have an associated driver containing an open, close, read, and write routine. The routines determine how the open, close, read, and write calls are handled. A device which operates with interrupts also contains a routine to determine how interrupts are handled. Additionally, there may be a special function routine in the driver which performs a particular action for that device. Other routines may appear in a driver but they are called out from within the open, read, write, or close routines.

In the foregoing discussion it is important to distinguish between call and routine. The call is user generated software with user supplied arguments. The routine is the system software residing in the driver which determines how the call is handled. The rules assigned to the use of the each call may be amended if the routine servicing the call is changed.

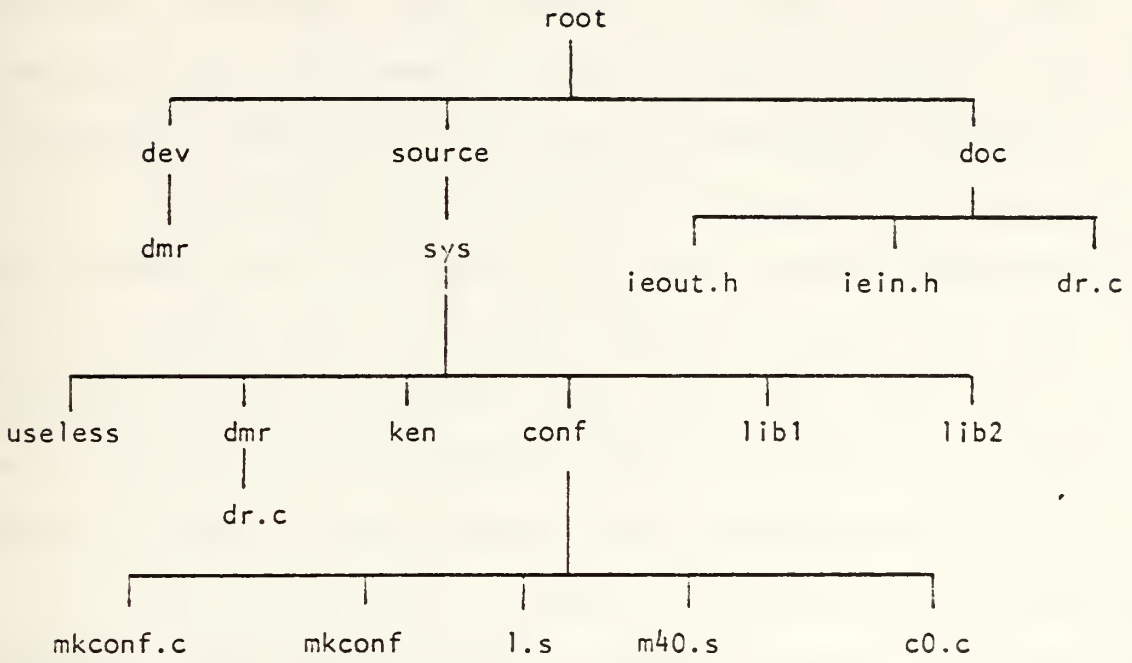
2. I/O Support Software

If the GPIB interface involved only the amending of an existing driver, little effort would be needed to edit the driver and reconfigure

the system. Unfortunately, the Unix operating system at the SATCOM laboratory has no installed software for such an interface. Therefore, one must be able to create a driver and then reconfigure the system to make the driver operational. Several files are involved with reconfiguration and are essentially independent of the form of the driver software, so long as the driver is written in the C language and consists of the routines previously mentioned. These files will be discussed here to simplify the description of the reconfiguration process which follows.

Filenames in Unix follow a few important rules. Files which are written in the C language which are to be compiled by the C compiler must have the form 'filename.c' while their compiled code takes on the form 'filename.o'. For the few assembly language programs the form is 'filename.s' for the source code and 'filename.o' again for the assembled version. Files with no suffix may be directories, executable programs, or text. The following paragraphs address the files which directly impact on I/O. Figure 4.1 is provided to assist the user in locating the files mentioned.

The 'dev' directory contains the Unix special files for I/O. Each device on the Unibus is associated with a special file. The listing of dev is different from all other directories, containing the major and minor device numbers associated with the device. Each device has its own major device number as designated in the 'c0.c' file discussed later. If one driver applies to several devices of the same type (such as teleypes) the minor device number is used to identify each member of that device type. The only way to make an entry into the



Note: Only files and directories under discussion are depicted above and represent only a small segment of the Unix operating system.

Fig 4.1 Unix I/O file Tree

directory 'dev' is through the mknod(VIII) command which requires the device numbers as part of its argument list. The files in the 'dev' directory are named when initiating the open call as argl in open(argl,arg2). For example, to open the write routine of tty1 call open(argl,2), where argl is the address of the pathname '/dev/tty1'.

The file 'dmr' contains all the source code for drivers which are currently used by the system. The source code for drivers which are not being used is kept in the file 'useless' which is in the same directory, 'sys', as 'dmr'. The source code does not directly participate in the reconfiguration process, but serves as a workbench area for changes in driver structure before compilation.

The directory 'ken' contains the source code for the Unix operating system with the exception of device drivers. The source code is all in the C language and when compiled is the contents of the file 'libl'.

The file 'libl' is one of the fundamental building blocks of reconfiguration of the Unix operating system. 'Libl' is a library of the compiled code in 'ken', and is therefore accessible only through the "ar" command which permits a user to add, delete, or tabulate the contents of the library.

The file 'lib2' is similar to 'libl' except that 'lib2' contains the the object code of the driver routines. The SATCOM version of 'lib2' contains object code of all the drivers available to Unix including not only the contents of 'dmr' but of 'useless' as well. There is no reason that the 'lib2' file could not be purged of the object code which is not applicable to the SATCOM lab, but demand on

memory space has not yet warranted such a house cleaning operation. The procedure to alter the contents of a library is different than that for a simple file. Library manipulation is discussed in the following paragraphs on system reconfiguration.

The file 'm40.s' is one of the few assembly language programs involved in the reconfiguration process. It supplies a machine language set of functions necessary for the reconfiguration of the operating system. The file 'm40.o' is the compiled object code which is actually used during system configuration.

The 'l.s' file is the other assembly language file involved in the system configuration process. 'L.s' supplies interrupt vector information for all the device drivers. The object code version, 'l.o', is used in the configuration process. The Unix reference manuals label the 'l.o' file as 'low.o' and the 'l.s' file as 'low.s'. Use of the 'l.s' file is discussed in detail in the following reconfiguration subsection.

The 'c0.c' file contains the tables which relate device numbers to the driver routines. The Unix manuals refer to this file as 'conf.c' which should not be confused with 'conf.h' which is a "header" file. There are two tables: one for block devices and one for character devices. The character table is of concern here, and it requires some explanation. The character table has five columns (plus a comment column). Each row of the table refers to a particular device driver. The entries in the row determine which routines are contained in the driver (open, close, read, write, special). If the driver has no open or close routine &nulldev is entered in that position of the row. If

the device is missing any other routine &nodev is entered ("&" means routine address). If there are no routines at all (the driver is not in use) &nodev is entered in all row positions. The row number, starting with zero, determines the device major device number. For example, the driver for the line printer (lp) is in the third row of the table, so it has major device number 2. It has no read or special routine (&nodev) and its remaining routines are lpopen, lpclose, and lpwrite.

The file 'mkconf.c' is the source code for the executable file 'mkconf'. The purpose of mkconf is to create the 'l.s' and 'c0.c' files. Use of this method to reconfigure the operating system is limited to the case where complete regeneration is required. As long as only one driver iseing added to the system, the easier tact is to edit the source codes 'l.s' and 'c0.c' and recompile the object code. A header to 'mkconf.c' should be added in comment form which explains that the program has be circumvented in the event that future system ammenders attempt to use an obsolete 'mkconf' program.

There are additional files which appear in I/O related programs and functions called header files. These files always are suffixed by the letter "h" (e.g., 'param.h') and are picked up via "include ..." as required by the program. Files 'param.h', 'user.h', 'conf.h', 'tty.h', and 'buf.h' are the more recurrent header files. The reader may peruse these at his convenience, but since they play no direct role in reconfiguration, no discussion of the header files appears here.

C. SYSTEM RECONFIGURATION

The following stepwise procedure to reconfigure the operating system is designed for installing a new I/O driver into the system as opposed

to amending an existing one. Modification, discussed in the next subsection, uses some of the following steps, and a knowledge of the full reconfiguration process will make the modification process easier to understand. Also, one must be aware that only the Super-User in Unix is allowed to alter the operating system. Some files will permit editing from any user, but most require the Super-User.

1. Driver Installation

The first step in reconfiguration is to develop the device driver software. This development for the GPIB interface driver is discussed in the next section. The driver need not have all the bugs worked out of it to be installed in the system, since the reconfiguration process only requires the driver to be compilable. The effectiveness of the routines within the driver will be tested after successfull reconfiguration and booting up of the new operating system. In this procedure we use the driver designation "xx" so that the C language program will be created under the filename 'xx.c' . Since the driver consists of a set of routines not attached to a "main" routine, it is not compiled as an executable program. In compiling the driver 'xx.c' use the following command:

```
cc -c -O xx.c
```

The output of this command is the compiled file, 'xx.o'. The two flags, "-c" and "-O", suppress the loading phase of compilation and optimize the object code, respectively. The 'xx.o' file is compiled but not executable.

Once a driver is ready for installation we need to assign it a major device number. Additionally, if the driver will service more than

one of the same type of device, minor device numbers are required. To obtain the major device number, edit the 'c0.c' file by inserting a row in the character table corresponding to the routines in the driver. Placement of the row is very important. As discussed in the above paragraph on the the 'c0.c' file, the major device number depends on the location of the row of driver information. If the xx driver information is placed above that of a driver being used by the system the major device number for the old driver will increment by one, but this will not be reflected in the 'dmr' directory. To avoid any problems place the driver information row below any currently used drivers or append it to the bottom of the table. The major device number is now the number of the row (the first row is number 0.)

Now that the driver's major device number is known we can enter the name of the driver into the 'dev' directory usin the mknod(VIII) command. Again, this command will work only for the Super-User. Mknod(VIII) has the form:

/etc/mknod xx c major minor

where xx is the name of our driver, "c" refers to a character device, and "major" is the major device number. If there are minor device numbers the mknod(VIII) command is repeated for each minor device number starting at zero.

The directory 'dev' reflects both the major and minor device numbers which are created by the mknod(VIII) procedure. One should check of the 'dev' directory to insure that the device name and associated major and minor numbers are indeed present in the system. Having taken care of the major and minor device number requirements we

now place the object code of the driver in 'lib2'. The means to make changes to a library or archive is through the "ar" command. Append the object code of the compiled driver, 'xx.o', to 'lib2' by executing the command:

```
ar r lib2 xx.o
```

Check the new contents of the library by typing

```
ar t lib2
```

which will list the contents of the library. The 'xx.o' file should appear at the end of the list of contents.

The entering of the driver's interrupt is not dependent on the successfull completion of all the former compilation steps. The file '1.s' may be changed even prior to writing the driver software. The step is placed here for convenience. The entrance of the the device interrupt vector involves placing a pointer to a callout routine and the device's priority level in the vector. The vector entries in '1.s' must be in order since the assembly language location counter may not be moved backwards. The Unix Assembler Manual [Ref. 7] expands on the restrictions of the notation in '1.s'. The second entry listed under the comment "interface to C" saves registers as required and makes a call on the driver's interrupt routine. When the routine returns, the registers are restored. The syntax of making the entries into the file '1.s' follows the same style as the other entries in the file. When the '1.s' file has been edited to satisfaction it should be assembled and the output moved to filename '1.o'.

At this point the driver has been successfully compiled into 'xx.o', the character confguration table in 'c0.c' has been changed and

compiled into ´c0.o´, the ´lib2´ file has had ´xx.o´ appended to it, and the ´l.s´ file reflects the proper interrupt vector and routines assembled into the However, prior to its use ensure that the argument files supplied to "ld" are in the working directory. If not, supply the full pathname of the file rather than its ordinary filename. Type the command:

ld -x 1.o m40.o c0.o libl lib2

to create a new operating system. The ´m40.o´ and ´libl´ files are unaltered versions of the previous system. the "-x" flag saves some space in the output file by preserving only external symbols. The result of the "ld" command (if no errors are encountered) is an executable ´a.out´ file, which may be moved to the name ´unix´ in the final form. If there is doubt as to the accuracy of ´a.out´, it may be wise to change the name to ´xunix´ to avoid confusion with the former system.

Once the new operating system, ´xunix´, has been created, copy it to the ´root´ directory. Once in the root, the system may be brought down and rebooted with ´xunix´ instead of the usual ´unix´. If all is well, and the old system is obsolete, rename ´xunix´ to ´unix´. The main reason for the renaming is that the command "ps" (process status) must work on the operating system named ´unix´, or else the attempt to execute the "ps" command results in a "no swap device" response.

2. Driver Modification

If an existing driver requires modification the process described above is quickly shortened. Following the same process, we need only edit the driver´s source code in the ´dmr´ directory (where it

was placed upon successfull system reconfiguration), and compile. Unless there is a change in the interrupt routine, no other source code need be altered. The changed driver is placed on the end of the 'lib2' file ("ar r xlib2 xx" will replace the old driver code) and the "ld" command is executed as above. The remaining file shuffling is the same as full reconfiguration.

The driver for the GPIB was not part of the original library or configuration table. Therefore, the full reconfiguration process was required for the interface implementation. The next section discusses the first step in that reconfiguration: development of the driver software routines.

V. SOFTWARE IMPLEMENTATION

Three essential areas to the development of the GPIB interface have been discussed: the characteristics of the General Purpose Interface Bus, the hardware design of the GPIB interface (DR11-C and CCI), and the I/O requirements of the Unix operating system. These three areas dictate the operational requirements of the software necessary to implement the interface. Effective software provides the user with the necessary functions to be able to read from or write to a device on the GPIB while working within the Unix operating system.

This section covers two areas of software development. The routines in the GPIB device driver/handler are discussed in the first subsection followed by a treatment of the two header files ('iein.h' and 'ieout.h') which act as software interfaces between the user's program and the driver. This report does not contain a printout of the software for any of the three functions. The software is available as files in the 'doc' directory mounted on disk seven of the workbench PDP-11/34A in the SATCOM lab. The source code for the driver is file 'dr.c', which also resides in the 'dmr' directory as discussed in section IV. The object code 'dr.o' is appended to 'lib2' in the 'sys' directory. The header files 'iein.h' and 'ieout.h' also reside in the 'doc' directory. They have no separate object code since they are compiled at the same time as the user program which includes them.

A. GPIB INTERFACE DRIVER

The GPIB driver is a file in the Unix system containing five routines: dropen(), drclose(), drwrite(), drread(), and drint(). Each of these routines is discussed separately. The set of driver routines is preceded by a set of constant definitions and structure assignments. The constant definitions take into account the fact that output from the DR11-C is inverted prior to being echoed back to the input register, while input from the GPIB is not inverted as it is passed to the input register of the DR11-C (see section III). The constants prepended with an 'X' are bit inverses of their namesakes without the prepended 'X' (e.g., XATN is the inverse of ATN). The structure system sets up the input and output register addresses of the DR11-C relative to the defined value for the control/status register.

1. Dropen()

The dropen() routine opens the GPIB driver for reading or writing. It first checks to see if anyone else has control of the driver by determining if "dr11.drstate" has a value other than zero. If it does, dropen() sets an error and returns to the user with a "-1" file descriptor. If "dr11.drstate" is zero, dropen() clears the control/status register which disables interrupts. An integer value for a file descriptor is returned to the user since no error bit was set by dropen(). The assignment of the value for the file descriptor is made by the system open call whose source code is located in the 'sys2.c' file in the 'ken' directory. Interrupts in the context of this driver are discussed in the paragraphs dealing with the drint() routine.

2. Drclose()

The drclose() routine is the simplest of the driver routines. It has one line of code which zeroes the "drll.drstate" to indicate that the user has no more use of the driver and others may access it.

3. Drwrite()

The drwrite() routine takes a character from the user's data buffer (called 'buf' in section IV) and places it into the output register of the DRll-C. It then intiates the handshaking process with the computer as the talker (see section II). The routine loops through this process until one of two terminating events occur: the number of characters specified by the user to be transferred to a GPIB device is exceeded, or the delimiting character is received signifying the end of the character string. The standard C language string delimiter is the ascii NULL character.

The system routine cpass() is used to control the looping and place the DATA in the DRll-C output register. Cpass() is described in the Unix I/O manual [Ref 10: pp. 2-3]. The source code for cpass() resides in the the 'subr.c' file of the 'ken' directory. The arguments of the system write call (section IV) are sensed by cpass() which uses the 'nbytes' argument to judge how many characters should be passed and counts down from 'nbytes' for each call on cpass(). As long as the loop number has not exceeded the 'nbytes' limit, the routine returns the character in the user's buffer ('buf' argument of the write call). If an error occurs or the count becomes zero, the value "-1" is returned.

The drwrite() routine uses the value returned by cpass() twice in each loop. Initially, cpass() takes a character from the user's

buffer and ensures that it is an ascii character. Therefore, no bit masking is required in drwrite(). If the value returned from cpass() is less than zero, there is nothing more to write and the drwrite() routine returns. The second use of the returned value from cpass() is to check for the NULL delimiter at the end of each loop. The loop in the drwrite() routine is of the do-while type to enable NULL to be placed on the GPIB to reset the DATA lines. If the character returned from the cpass() routine is valid it is placed on the DR-11C output buffer and the handshaking sequence is initiated.

The GPIB handshaking in software is accomplished through a series of bit checking and bit setting in the DR11-C input and output registers, respectively. After the DATA is placed on the DATA lines of the output register, the talker (computer) tests the NRFD bit in the input register until NRFD is high. This signifies the readiness of all listeners to receive data. The DAV line of the output register is then set high, which, through the inverters, is echoed to the input register as DAV low (true: DATA is valid). Once DAV is made true on the GPIB, drwrite() waits for NDAC to become high in the same manner as it waited for NRFD. NDAC high signifies that all the listeners have accepted the DATA, so the talker sets DAV high, places a new character on the DATA lines, and starts the cycle over.

The method of bit setting in the input register is somewhat complex due to the inverters between the input and output registers. To set a bit low in the input register, the contents of the output register are inclusive OR'd with the defined constant representing the GPIB signal line to be set (e.g., DAV). To set the bit high in the input

register, the contents of the output register are AND'd with the inverse of the constant (e.g., XDAV).

If the user attempts to write to a device on the GPIB which does not exist there is no effect on the workings of drwrite(). The wired AND characteristic of the GPIB (see section II) allows all the listeners to respond to the talker, preventing the loop from hanging up in an attempt to sense a signal which never comes.

The last action of the drwrite() routine is a test of the returned value of cpass() to determine if the character returned is the NULL delimiter. If it is not, then the loop repeats.

4. Drread()

The drread() routine enables a user to read a string of ascii characters from a device on the GPIB. The routine transfers characters until the ascii character Line Feed is detected as DATA, or the EOI (End Or Identify) bit of the GPIB is set true in the data mode (ATN false). The user may terminate the number of charaters read before one of these two conditions is met by the selection of an appropriate integer for the argument 'nbytes' in the system read call. Section IV provides a full discussion of the read call and the argument 'nbytes' used by that call.

Drread() uses the system routine passc(c), the companion of cpass() discussed above. Passc(c) is used in drread() in a similar manner as cpass() is used in drwrite(), except that passc(c) takes data from the DR11-C input register and passes it back to the user's buffer named by 'buf' in the read system call. Passc(c) monitors the number of characters transferred in the same way as cpass() but does not return the character transferred as cpass() does. Rather, it returns the value

"0" until it's counter determines that 'nbytes' have been passed at which time it returns "-1" as a flag.

The drread() routine uses a do-while loop similar to that of drwrite(). Before entering the loop the computer (as active controller) sets NRFD and NDAC both true and resets the DATA lines false. Then ATN is set false, which tells the talker to commence sending data. The listener/talker relationship is established prior to the call of drread() through the drwrite() routine (see the discussion of 'iein.h' below).

The processing of data within the drread() loop is more complicated than that for the drwrite(). In drwrite() the cpass() routine took care of masking off the ascii characters. Passc(c) does not. Second, the input from the GPIB device is negative-true which is not inverted prior to entering the DR11-C input register. Third, a means must be supplied to check to see if the addressed device to supply the data is in fact on the bus. An infinite waiting loop is possible and must be avoided; yet some devices may be slow to deliver data. Therefore, a limit to the time for a device to supply data must be set. Fourth, in the read mode the computer is the listener and must supply two handshake signals (NRFD and NDAC) rather than one (DAV).

On the first pass through the do-while loop, the drread() routine checks to determine if the device to supply data is on the bus. Using LCNT as a counter in a bit checking loop, the computer waits for the DAV line to be set low by the talker after setting the NRFD line high (to signal that the computer is ready for DATA). This loop is repeated 3000 times or until DAV is sensed low, whichever comes first.

The value 3000 results in a time interval of approximately 100 milliseconds and was determined through trial and error experiment using a HP-1615A Logic Analyzer. If DAV is not detected in 100 milliseconds, the goof() routine is called to send the character "?" to the user's buffer and then return.

If DAV is sensed low (true) drwrite() sets NRFD high (until it is finished reading in the DATA) and passes through a second check for DAV (used for subsequent loops). It then processes the DATA on the input register DATA lines to the user's buffer. The first process is the check for a delimiter from the talker. If the Line Feed ascii character or the GPIB End Or Identify bit is detected, the routine jumps to the the label "out" which supplies the NDAC handshake, sends the user's buffer the delimiter NULL, and resets the bus prior to return. If the delimiters are not sensed the 16 bit word on the input register is masked off to eight and inverted. The passc(c) routine then sends the processed data to the user's buffer. After sending the DATA to the user's buffer, drwrite() sets NDAC high, which informs the talker that DATA has been accepted. After the talker recognizes NDAC high and sets DAV high, NDAC is set back to low. The loop in drread() is repeated until the user's number of characters are passed to his buffer, or one of the delimiters (Line Feed or EOI) is received from the device supplying data.

5. Drint()

There is no routine for interrupts for the GPIB driver. All exchanges of DATA and handshaking are performed by bit setting and checking in the DR11-C output and input registers. The nature of an

interrupt on the GPIB is in the form of the SRQ signal which is designed to inform the system controller of a situation in one of the devices on the bus which requires service, such as a printer out of paper. The SRQ signal is only a request to the system controller who must poll the bus, serially or in parallel, to find out which device set SRQ true. A routine to check SRQ is not included in the current driver implementation since the read and write routines are not adversely effected by it. Subsequent refinements of the driver may contain such a status checking routine as a special function. (see section XI).

Since no interrupts are used during a read or write transaction in the driver, the interrupt enabling bits in the DR11-C control/status register are disabled by clearing the register in the dropen() routine. The CONCLUSION section discusses possible alternatives to the above method of reading and writing which uses the concept of system interrupts in the handshaking process.

B. GPIB DRIVER SUPPORT SOFTWARE

The user has two header files, 'ieout.h' and 'iein.h', available to assist in the transfer of data between the Unibus and the GPIB. Use of the ieout and iein routines contained in these files eliminates the need for the user to compound his program with separate software to set up the talker/listener relationship required by the GPIB. Header files 'iein.h' and 'ieout.h' are appended to the user's source code (after the 'main' routine) via the 'include' method. Figure 5.1 is a simple source code for writing the string "12345" to a GPIB printer showing the use of the header file 'ieout.h' in the 'include' format. Note, header files must begin with the "#" character.

```
#

main()
{
  int d,L;
  char *info;

  L = 8;                    /* length of info to write is 8 */
  info = "12345\r\n";       /* data to write is 12345 */
  d = 06;                   /* device number is 06 */

  ieout(d,L,info);          /* write out the info */
}

#include "../ieout.h"
```

Fig. 5.1 Sample Program To Write To a GPIB Device

1. Ieout.h

The output support file, 'ieout.h', contains the ieout routine which sets up the computer as the talker and the GPIB device designated by the user as the listener. Ieout is of the form ieout(device,length,array) where the argument 'device' is the listener device number expressed as an octal number, the argument 'length' is the length of the user's string to be transferred (including the implied NULL delimiter), and 'array' is the name of the string to be written.

Ieout contains two write system calls. The first sends two bus commands to set up the talker/listener relationship following the UNLISTEN command. The computer is designated as talker and given talk address code and may be changed to another value using the "Ascii Character Set." [Ref. 3: pp. 75,76] Ascii 'U' should be avoided since that address is supplied by Hewlett-Packard as the address for the HP-98034A Interface for the Hewlett-Packard desk top computer, HP-9825. [Ref 4: p. 12] The listener address consists of the argument 'device' supplied by the user OR'd with the basic listener designation code from the ascii character set.

The second write system call in ieout sends the user's DATA to the bus. In both write calls the drwrite() routine in the GPIB driver takes the second argument to be the the address of the array or string to be transferred. The remainder of the code in the ieout routine supplies the open, close, and file descriptor information required to write out DATA. These system calls are discussed in section II.

Figure 5.1 illustrates the use of ieout in a user program. Execution of the program prints the number "12345" on the HP-7245B

printer/plotter whose device number is 06. If the value of 'L' in the program was larger than 8 (the number of characters in the "info" string) the printout would still be the same since the C language supplies the NULL delimiter automatically at the end of a string. If the value of 'L' is smaller than 8, only the designated number of characters are sent to the bus. Therefore, 'L' is recommended to be assigned an integer value larger than the number of characters in the string.

2. Iein.h

The input support file, 'iein.h', contains the iein routine which sets up the talker/listener relationship in the same way as ieout, except that the computer is designated as listener and the device to supply DATA is designated as the talker. The computer listener address is directly related to its talker address discussed above. The basic address for both cases is the five least significant bits of data on the GPIB (with ATN true). The sixth and seventh bits determine whether the five bits refer to a talker address or a listener address [Ref 3: pp. 52-53]. Iein is of the form iein(device,length,array) where the 'device' argument is used in the same way as it was in the ieout routine. The 'length' argument is used for the same reason as the 'length' argument in ieout. Iein uses the drread() routine as discussed in the drread() section.

The function of the iein routine is similar to ieout in that it writes out the talker/listener relationship and then reads in the data into the user's buffer. Ieout writes out the talker/listener relationship and the contents of the user's buffer to the bus.

Therefore, the two support routines, ieout and iein, provide the user with installed software to handle the talker/listener relationship assignments. The user need only supply the required arguments to the routines.

Figure 5.2 is a sample program that reads in data from device number 01 into the array "store" and subsequently prints out the contents of "store" onto the console screen using the "printf" call. In order to read data, both the iein and ieout routines are used. First, ieout is used to tell device number 01 to supply data from its channel B. Iein then sets up the talker/listener relationship and receives the DATA. As with the ieout routine, the user should provide a large enough integer for the 'length' argument to cover the amount of DATA anticipated from the talker device. If 'length' is larger than the desired number the delimiters from the talker control the loop in drread(). All instrumentation on the bus use some form of ascii Line Feed as a delimiter (some in combination with an ascii Carriage Return) so no errors are supplied to the user in his buffer. Finally, since both iein and ieout are used in the user's program, their respective header files, 'iein.h' and 'ieout.h', are appended to the main routine of the user program.

C. SUMMARY

Three software routines implement the interface between the Unibus and the GPIB. These routines are based on the requirements of the Unix operating system, the GPIB bus, and the hardware used to interconnect

```
#

main()
{
  char *setdev, *store;

  setdev = "BI\r\n";     /* message to set up
                            device supplying store */
  ieout(01,4,setdev);    /* write message to device 01 */

  iein(01,12,store);     /* read 12 chars into "store" buffer */

  printf("%s",store);    /* print "store" on console */

}

#include "iein.h"
#include "ieout.h"
```

Fig 5.2 Sample Program To Read From a GPIB Device

two bus systems. The three software routines are the GPIB driver and two support routines, ieout and iein, which interface the user's software and the GPIB driver.

The GPIB driver is a set of five routines with drint() remaining void but available to be filled in the event that the driver is changed to an interrupt driven handler. The current driver makes no use of the interrupt routine since the hardware interface was not designed for interrupt implementation. All DATA transfer and handshaking is accomplished through bit setting and bit checking in the DR11-C output and input registers.

Two user available header files are supplied to assist the user in reading from or writing to a GPIB device. These two files contain software which deletes the need for user generated code to handle talker/listener relationships required by the GPIB.

VI. CONCLUSIONS

The interface between the Unibus and the General Purpose Interface Bus consists of two hardware modules, the DR11-C and CCI, supported in software by the GPIB driver and two support routines contained in the header files 'ieout.h' and 'iein.h'. This hardware and software system effectively enables a user to write to or read from a device on the GPIB using C language software on the Unix operating system. The following subsections discuss the advantages and disadvantages of the installed interface plus a comparison of it to a commercially available product.

A. ADVANTAGES

One advantage of the GPIB interface is its hardware simplicity. The DR11-C is a readily available interface to the Unibus which requires no hardware or software changes beyond the register and vector address wire jumper connections. The CCI board is a channeling device with a set of inverters between the DR11-C output register and the GPIB. One driving force in insuring simplicity of hardware is the limited physical space available to the CCI. Since the entire interface structure is constrained to one hex-height equivalent card in the PDP-11 mainframe, and the DR11-C takes up two-thirds of that room, the CCI is limited to an effective area of a 5 by 7 inch card.

The software to support the hardware is also simple. Developed under the requirements of the Unix I/O system, the GPIB driver conducts all read and write data transfer, including bus commands and handshaking between the input and output registers of the DR11-C. There are no

complicated jumps to interrupt routines or need to put the system to sleep waiting for input or a handshake response.

The supporting "iein" and "ieout" routines available for the user relieve any programming requirements specific to the GPIB. The user need only supply the number of the device to receive or supply data, the amount of data for transfer, and the buffer from which or into which the data flows. The supporting routines handle the GPIB talker/listener assignments and manipulate the system open, close, read, and write routines.

B. DISADVANTAGES

Transfer of data on the GPIB is asynchronous and inherently slow. The handshaking process is only as fast as the slowest device on the bus. Under the current system of handshaking through the DR11-C input and output registers, the Unix operating system can become bogged down waiting for data transfer to flow over the GPIB. Since Unix is a multi-user system, the delay caused by the slow GPIB can cause inconvenience to other users on the system.

One possible alternative to correct the slowness problem is to run the handshaking activity through an interrupt system. Such a system will require rewiring of the CCI since there are currently no connections to the interrupt lines of the DR11-C control/status register. Also, the software will require modification to enable the DAV, NRFD, and NDAC signals to effect the control/status register. Though the software and hardware will become more complex, the tradeoff for a faster system may be worthwhile if the use of the GPIB is extensive. Since some devices on the GPIB have relatively slow data

rates (e.g., the printer/plotter), use of the noninterrupt driven interface is optimum when multi-user demands are minimum, or when only one user is on the system.

Under the current implementation the system controller and active controller are permanently designated as the computer (Unibus). There is no provision for the passing of control to another device on the bus. However, the computer acts as a nonaddressable listener if a GPIB desk top computer (such as the HP-9825) takes over the bus when no users are transferring data. The hardware and software of the GPIB interface leave the REN line of the GPIB in the true position so that if another computer is attached to the bus, it need not place the other devices in remote.

Lastly, the current implementation is not able to respond to a service request by a device on the bus. There is no provision for the use of a serial poll or parallel poll to determine the source of the request. This deficiency could be corrected through a software routine appended to the driver as a special function routine to determine the status of the bus.

C. COMPARISON WITH A COMMERCIAL PRODUCT

Not surprisingly, there are commercially available products to interface the Unibus and the GPIB. One of these, the GPIB11-1, is manufactured by the National Instruments Company of Austin, Texas. As one may expect, the GPIB11-1 is promoted as a very effective and efficient interface, clearly outperforming the GPIB interface discussed in this report. The notable differences between the two interfaces are discussed below. In order to compare and contrast the two interface

systems the following definitions apply: the term "GPIB11-1" refers to commercial product, while "the GPIB interface" refers to the subject of this report.

The GPIB11-1 hardware is the same size as the DR11-C alone, a single quad height board which plugs into a slot in the PDP-11 mainframe. The GPIB interface hardware takes up the equivalent of a hex-height board.

The GPIB11-1 software includes a Utility program and an Interactive Control program in addition to the Driver program containing C callable subroutines. The Unix driver is not standard equipment. However, it is available as an option. The key differences in software are that the GPIB11-1 is interrupt driven and will interrupt the PDP-11 when:

(1) the GPIB11-1 is the talker and is ready to send data,
(2) the GPIB11-1 is a listener which has received data,
(3) the GPIB11-1 is an active controller and a device on the bus has set SRQ true,
(4) the GPIB11-1 is not the system controller and the interface is being cleared, or
(5) the GPIB11-1 is a bus monitor and a command byte has been received.

The GPIB11-1 is capable of passing control of the bus to any other device on the bus. The GPIB interface maintains control at all times.

The GPIB11-1 is capable of servicing a SRQ signal from any device on the bus via a parallel poll to determine device status. The GPIB interface does not service SRQ.

The above differences do not list all the functions of National Instruments' product. A full set of hardware and software specifications are available in the National Instruments product specification sheet on the GPIB11-1 [Ref. 12].

There is an additional important difference between the GPIB11-1 and the GPIB interface: cost. With the optional Unix driver software the

GPIB11-1 costs $1895.00. A version which includes DMA capablities costs $2695.00. These are costs to purchase one unit. The cost conceivably could be lower if larger quantities are purchased. The DR11-C can be purchased for $400.00 while the CCI is locally made. A straight dollars and cents comparison of the two systems is somewhat misleading, since a true cost comparison must include time and labor spent in the development of the device and software.

If a decision is required as to the most cost effective interface to use in a Unix system, careful analysis of the needs of the system is necessary. There is no current need in the SATCOM lab to pass control of the GPIB to any other device on the bus. Under the current system there is no need to control the flow of data via an interrupt method, unless significant inconvenience to other users is experienced. Even if such need for interrupts arises, the current system can be hardware and software modified to fulfil the requirement. If a need arises to poll the devices on the bus as a result of a service request (SRQ set true), the GPIB driver can be adapted with a special function routine to obtain the status of the devices on the bus.

D. RECOMMENDATIONS

The GPIB interface is recommended as the means to transfer data between the Unibus and the GPIB under the control of the Unix operating system. The additional features of interrupt control of the data exchange process and a polling routine for service requests may be incorporated into the system without purchasing a commercial interface. The addition of the special function to poll for the device setting SRQ may be accomplished without change to CCI hardware, since the SRQ line

is monitored by the input register of the DR11-C. Changing the format of GPIB handshaking to interrupt control may create a more convenient system, but it will be faster only during times that data actually flows to or from the GPIB and more than one user is on the system. The exchange of data on the GPIB can only proceed as fast as the slowest listner, despite the method used to implement the handshaking. Changing over to an interrupt driven system will require hardware changes to the CCI since the control/status register of the DR11-C has no input from the current CCI design.

It is recommended that the installed system be incorporated as is for a test period of time to be determined. During the test period the need for changes as discussed above may be evaluated.

LIST OF REFERENCES

1. Digital Equipment Corporation, (no title), "PDP-11/34 Handout," (no date).

2. MDB Systems, Inc., DR11-C General Purpose Interface Instruction Manual, 1976.

3. Hewlett-Packard Company, Interfacing Concepts and the 9825A, part no. 09825-90060, 1976.

4. Hewlett-Packard Company, Hewlett-Packard 98034A HP-IB Interface Installation and Service Manual, part no. 98034-90000, 1979.

5. Naval Post Graduate School Satellite Communications Laboratory, IEEE 488 Bus Interface Board Schematic, drawing no. CCI-C-00, 1981.

6. Bell Laboratories, The Unix Time Sharing System, by D. M. Ritchie and K. Thompson, 1974.

7. Bell Laboratories, Unix Assembler Reference Manual, by D. M. Ritchie, (no date).

8. Bell Laboratories, Programming in C - A Tutorial, by B. W. Kernighan, (no date).

9. Kernighan, B. W. and Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

10. Bell Laboratories, The Unix I/O System, by D. M. Ritchie, (no date).

11. National Instruments Corporation, GPIB11-1 PDP-11 Unibus Interface to IEEE Standard 488-1975 Instrumentation Bus, (product specification), 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 62 Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	1
4.	Kenneth G. Gray Code 62Gy Naval Postgraduate School Monterey, California 93940	5
5.	LCDR Ayers H. Blocher III 11382 Cromwell Court Woodbridge, Virginia 22192	1

Thesis
B566 Blocher
c.1 Hardware and soft-
ware implementation of
an interface between
the unibus and general
interface bus.